Journal of Big Data

RESEARCH **Open Access**

CoPart: a context-based partitioning technique for big data

Sara Migliorini*, Alberto Belussi, Elisa Quintarelli and Damiano Carra

*Correspondence: sara.migliorini@univr.it
Department of Computer Science, University of Verona, Verona, Italy

Abstract

The MapReduce programming paradigm is frequently used in order to process and analyse a huge amount of data. This paradigm relies on the ability to apply the same operation in parallel on independent chunks of data. The consequence is that the overall performances greatly depend on the way data are partitioned among the various computation nodes. The default partitioning technique, provided by systems like Hadoop or Spark, basically performs a random subdivision of the input records, without considering the nature and correlation between them. Even if such approach can be appropriate in the simplest case where all the input records have to be always analyzed, it becomes a limit for sophisticated analyses, in which correlations between records can be exploited to preliminarily prune unnecessary computations. In this paper we design a context-based multi-dimensional partitioning technique, called CoPart, which takes care of data correlation in order to determine how records are subdivided between splits (i.e., units of work assigned to a computation node). More specifically, it considers not only the correlation of data w.r.t. contextual attributes, but also the distribution of each contextual dimension in the dataset. We experimentally compare our approach with existing ones, considering both quality criteria and the query execution times.

Keywords: Big data, Partitioning, Technique, Context-based queries

Introduction

The need to process and analyze the overwhelming flow of data, due to the rise of social media, Internet of Things (IoT) and multimedia, has motivated the study and development of parallel data processing systems able to deal with them. However, the development of tools and technologies to efficiently process and share such data still poses open issues [19, 30]. In particular, MapReduce frameworks, like Hadoop [1] and Spark [2], rely on a programming paradigm that works with datasets divided into independent chunks, or *splits*. The main underlying assumption is that splits can be processed in parallel to produce partial results, which are combined from time to time, until the final result is obtained. Such an approach has been originally developed for bulk analysis, i.e., analysis that involves all the records, considering that the processing time for each record is approximately similar. This translates into a default partitioning technique that considers only the amount of bytes as splitting criteria: records are placed inside the same split until a predefined byte threshold is reached, without inspecting its actual content.

In recent years, different specialized processing tools have been developed starting from these general purpose systems. An example is SpatialHadoop [17], which extends Hadoop by adding spatial operations and analysis, with queries based on data attributes, like time intervals or spatial regions [11, 25]. These queries are usually selective, i.e., they work on a portion of the data. A simple partitioning based solely on the size of the split may not be efficient, because it does not take into account the correlation of data.

Consider for instance an application scenario related to the tourism, with a dataset that contains the visits of tourists at different Points of Interests (PoIs). The tourists have a city pass which they swipe at the entrance of a PoI. Each swipe contains the identifier of the city pass, the name and location (coordinates) of the visited PoI, together with an entrance timestamp. If someone wants to analyze the dataset from a global point of view, e.g. to determine tourism trends, she can compose queries by considering the timestamp (How many tourists have been there in a specific day?), or the space (How many times has a specific PoI been visited?), or PoI type (Are modern-art museums preferred to science museums?). Complex queries may also combine different dimensions (How many tourists visited a specific PoI in a specific hour?) or types of analysis (Which are the PoIs close to my current location? What can I visit in a one-day trip?). In the above query examples, the analysis is based on some selective predicates: if the partitioning technique is able to help in pruning unnecessary data as soon as possible, the overall performance would increase, since it would be possible to balance the amount of parallel work.

In this paper, we use the term *context of analysis* to identify the set of dimensions (attributes) used to analyze a dataset. Our aim is to design a *context-based* partitioning technique [26] which tries to produce splits containing only context-related [12] records. Indeed, it has been recognized that the notion of context can be used to extract and present the relevant chunks of knowledge, thus allowing for information focusing and reduction [13]. The main aim is to implement a partitioning technique that, given a query based on the context attributes, is able to prune away uninteresting data without processing them. "Motivating example" section illustrates an example in which the use of such technique will significantly improve the performances of analysis operations. Clearly the selection of the context for the analysis, i.e., the identification of the attributes of interest, is a challenging task, which may have a great impact on the partitioning result and on the performances of the analysis. A possibile solution is based on the fact that some analysis are more frequent than others: by mining frequent attributes mentioned in logs of past queries, along with the knowledge of the target scenario, one can infer the most useful context. Once the context of analysis has been identified, another important issue is the identification of the boundaries of each k-dimensional split, i.e., the way data have to be grouped with respect to each contextual dimension. As discussed in "Related work" section, available techniques usually rely on a uniform space partition. However, this choice could be harmful in the case of context dimensions not uniformly distributed. Some task may end up with little or no work, while others could be overloaded [9, 10], which affect the benefits of a parallel computation.

The contribution of this paper is the definition of a context-based multi-dimensional partitioning technique, called CoPart, that, given a dataset D and a set of k contextual dimensions relevant for the analysis, is able to produce the most appropriate partition of *D*. The resulting splits not only contain context-related records, but they are also balanced, since during the partitioning the distribution of data related to each contextual dimension inside *D* is considered.

In evaluating our solution, we start from the tourism scenario described above, i.e., a real-world dataset containing the swipes of a city pass of an Italian city. Such a scenario is characterized by recurrent queries performing the same type of analysis, and the partitioned dataset is stored permanently on a distributed filesystem (e.g., on HDFS). The solution can be easily adapted to a dynamic scenario, where the dataset is kept in-memory (e.g., Spark), and repartitioned using a different context based on the current set of queries. In our experiments, we compare the partitioning produced by CoPart with the default one produced by Hadoop (baseline) and another technique available in literature. Such comparison is performed in two ways: (1) by defining and using some meaningful quality metrics, and (2) by collecting some experimental results on range queries. The results show that CoPart is able to provide better performances w.r.t. existing techniques considering both criteria.

The remainder of the paper is organized as follows: "Preliminaries" section introduces some basic notions that are useful for understanding the paper contribution, "Approaches to context-based partitioning" section discusses in details the ideas underling the proposed partitioning technique, "CoPart partitioning technique" section presents the CoPart partitioning technique, "Case study evaluation" section illustrates some experimental results confirming the goodness of the proposed approach w.r.t. other existing techniques. Finally, "Related work" section discusses some related work and "Conclusion" section summarizes the obtained results and proposes some future work.

Motivating example

Let us consider the tourist scenario described in "Introduction" section and its dataset *D*, whose records have the following structure: ⟨timestamp, latitude, longitude, cardId, ⟨timestamp, latitude, longitude, cardId, poiName, touristAge⟩.

Assuming that records are added to the dataset in a chronological order, as depicted in Fig. 1, when we apply the default partitioning technique available in Hadoop, we essentially subdivide the original file into *n* parts: scanning sequentially this file, records are placed inside the same split until a given threshold in bytes is reached (see Fig. 1). In the tourist scenario what we obtain is that each split contains contiguous records in terms of temporal attribute, but with very different values for all the other attributes. If we need to determine the average age of the tourists visiting the Arena, we need to process all the splits to identify the involved records and perform some operations on them, even if only 2 splits actually contain data related to Arena. Clearly, this is a toy example, but in real senario involving big datasets, where the number of splits exceeds the number of available nodes, the possibility of reducing the number of splits to be considered through a pruning technique, allows to reduce the number of sequential runs that have to be performed.

Moreover, since the number of records in each split related to the Arena could be very different, i.e. some splits can contain a huge amount of them (i.e., split$_3$), while

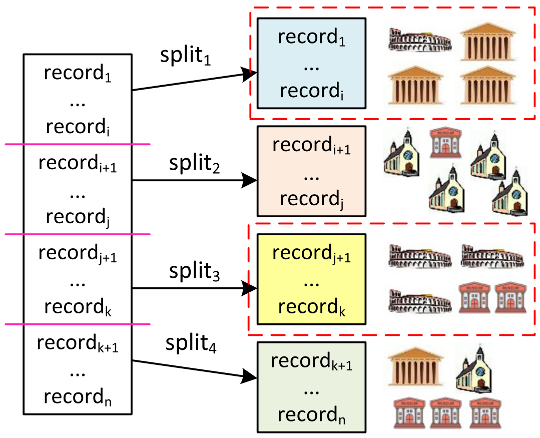

Fig. 1 Example of the default partitioning performed by Hadoop-like systems

other can contain very few of them (i.e., split$_1$), we obtain that the two instantiated map tasks will have a very different work to do. Indeed, each parallel task has not only to scan all the records in the split, but has to perform a more or less complex computation on those of interest. Since in a parallel computation, the overall duration is given by the duration of the slower task, for improving the global performance it is necessary to prevent the presence of slower tasks, namely to balance the amount of work to be done by each of them [23].

Preliminaries

In this section we provide a set of preliminary definitions as formalization of the context-aware partitioning problem. We start with the notion of dataset schema, from which a context of interest can be isolated.

Definition 1 (*Dataset*) A *dataset schema* $S = \langle a_1, \ldots, a_m \rangle$ is a list of attributes, each one belonging to a particular domain, denoted as $\Delta(a_i)$. A *dataset* $D = \{r_1, \ldots, r_n\}$ over a schema S is a collection of records $r_i = \langle v_{i_1}, \ldots, v_{i_m} \rangle$, where $\forall j \in \{1, \ldots, m\}\ v_{i_j} \in \Delta(a_j)$.

Let us consider the dataset D mentioned in "Motivating example" section which contains the visits of tourists at different Points of Interest (PoIs). The dataset D is characterized by the following schema $S = \langle \mathsf{timestamp}, \mathsf{latitude}, \mathsf{longitude}, \mathsf{cardId}, \mathsf{poiName}, \mathsf{touristAge} \rangle$, where timestamp and age are integers, while latitude and longitude are real values, and finally cardId and poiName are strings.

Given a record r, its component values are denoted with the following notation: $r_i[a_j] = v_{i_j}$ and $r_i[a_j, a_k] = \langle v_{i_j}, v_{i_k} \rangle$. The component values of r_i on a subset $S_k \subseteq S$ of the schema S can be specified without listing all the attributes, i.e. as $r_i[S_k]$. For instance, in the considered case $r_i[\mathsf{age}]$ is the age value for the record r_i, while $r_i[\mathsf{latitude}, \mathsf{longitude}]$ is a pair containing the values of latitude and longitude in r_i.

Definition 2 (*Context*) Given a dataset D over a schema $S = \langle a_1, \ldots, a_m \rangle$, a context of analysis is a subset of the attributes in S:

$$C = \{c_1, \ldots, c_k\} \subseteq S \tag{1}$$

Each attribute composing the context of analysis is also referred to as *dimension of analysis*. In a similar way a context defines a k-dimensional space on which the dataset records are distributed.

With reference to the motivating example, different contexts of analysis can be defined, for instance in order to determine tourism trends we can identify a context $C_1 = \{\mathsf{timestamp}\}$ referring only to the entrance timestamp, or we can consider space and time together $C_2 = \{$ timestamp, latitude, longitude$\}$, or in general we can consider complex contexts of analysis composed of many dimensions.

Given a dataset D, a generic partitioning operation produces a division of its records into a set of splits.

Definition 3 (*Partitioning*) Given a dataset $D = \{r_1, \ldots, r_n\}$, a partitioning P is a collection of subsets of D:

$$\begin{aligned} &P = \{p_1, \ldots, p_h\} \text{ such that} \\ &\quad \forall p_i \in P\,(p_i \subseteq D) \wedge \\ &\quad D = \cup_i p_i \end{aligned} \tag{2}$$

Each subset p_i is called *split*.

Given a context $C = \{c_1, \ldots, c_k\}$ defining a k-dimensional space, each record r_j inside a split p_i can be considered a k-dimensional point $pt_j = (r_{j_1}, \ldots, r_{j_k})$. Therefore, a split p_i could be intended as a set of k-dimensional points. Moreover, p_i covers a portion of the k-dimensional space defined by C, such portion of space is called k-dimensional region defined by p_i. This region can be manipulated by means of the classical spatial operations and the topological relations defined by the Point-set topology [15] can be applied to it.

In order to clarify this concept, take for instance a context $C = \{\mathsf{timestamp}, \mathsf{touristAge}\}$, each record $r \in D$ can be represented as a 2 dimensional point, as exemplified in Fig. 2. On such space, we can define a subdivision through a uniform grid. Each cell of the grid identifies a subset of D and the collection of all these subsets represents a partitioning of D.

Definition 4 (*Minimum Bounding Volume*) Given a set of records R with schema S and a context $C = \{c_1, \ldots, c_k\} \subseteq S$, the *minimum bounding volume* of R ($\mathsf{MBV(R)}$) is defined as is the minimum k-dimensional cube enclosing all the records in R.

Given a set R of records $\{r_1, \ldots, r_n\}$ and a context $C = \{c_1, \ldots, c_k\}$, the $\mathsf{MBV(R)}$ is determined by computing for each context attribute $c_h \in C$ its minimum and maximal values in R, denoted as $c_{h_{min}}$ and $c_{h_{max}}$ where:

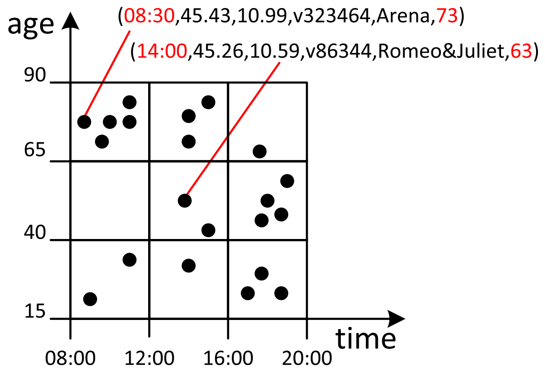

Fig. 2 Example of dataset represented as a set of points in a *k*-dimensional space

$$\begin{aligned} c_{h_{min}} &= min\{r_1[c_h], \ldots, r_n[c_h]\} \\ c_{h_{max}} &= max\{r_1[c_h], \ldots, r_n[c_h]\} \end{aligned}$$

The MBV(R) can be compactly represented by the tuple $\langle c_{1_{\min}}, \ldots, c_{k_{\min}}, c_{1_{\max}}, \ldots c_{k_{\max}} \rangle$.

In case the set R is a split p_j containing the records $\{r_1, \ldots, r_n\}$, the MBV(R) is the MBV enclosing the region of space containing the k-dimensional points of p_j.

Let us consider again the example depicted in Fig. 2, the MBV of the entire dataset is represented by $\langle 08.00, 15, 20.00, 90 \rangle$, while the MBV of the left-bottom cell is $\langle 08.00, 15, 12.00, 40 \rangle$.

Several different partitioning techniques can be defined on a dataset D w.r.t. to a context C. In particular, we can distinguish between *disjoint* and *overlapping* partitioning. In the first case, each record $r_i \in D$ can be placed inside one and only one split $p_j \in P$. Conversely, in the second case, a record r_i can be placed in more than one split $p_j, \ldots, p_k \in P$. As we can see in the following, in the case of overlapping partitioning, when a record is placed in more than one split, such splits have adjacent (or overlapping) MBVs.

Among all the possible partitioning approaches which can be defined on a dataset D, in this paper we are interested in those that are context-aware, namely, that place inside the same split records having a correlation w.r.t. the context attributes.

Definition 5 (*Context-aware partitioning*) Let us consider a dataset D with schema S and a context $C = \{c_1, \ldots, c_k\} \subseteq S$. A partitioning $P = \{p_1, \ldots, p_h\}$ for D is said to be *context-aware* with respect to a context C, if it defines a spatial subdivision of the k-dimensional space covered by D. Such subdivision is composed of h k-dimensional regions, each one associated to one split of P. Moreover, it stores inside the same split p_i only records r_j that have a not empty intersection with the k-region associated with p_i, i.e. only records r_j for which the associated k-dimensional point $pt_k(r_j) = (r_j[c_1], \ldots, r_j[c_k])$ is spatially contained in the k-dimensional region of p_i. Of course, the spatial union of the regions covers the whole reference space.

More specifically, we consider partitioning where the k-dimensional region (or simply k-region) associated to each split p_i can be represented by a MBV that draws the region boundaries, i.e. k-region(p_i)=$\langle c_{1_{\min}}, \ldots, c_{k_{\min}}, c_{1_{\max}}, \ldots c_{k_{\max}} \rangle$.

Given a record r and a split p_i:

$$r \in p_i \iff pt_k(r)\mathsf{intersects}(k\text{-region}(p_i)) \tag{3}$$

where the predicate $\mathsf{intersects}$ returns true if $pt_k(r)$ is spatially contained in the k-region of p_i, false otherwise.

If we consider again the situation depicted in Fig. 2, the chosen context is composed of two attributes (time and age) and it allows to subdivide the space into a set of 2D cells. Records are represented as 2D points inside such space, where their coordinates are given by the value of the attributes timestamp and touristAge. Each record is then associated to the cell which spatially contains its corresponding point. For instance, the record $r = \langle \mathbf{08}.30, 45.43, 10.99, \text{v323464}, \text{Arena}, \mathbf{73}\rangle$ has been associated to the cell in the first column and third row, since it corresponds to the time interval 08.00–12.00 and the age interval 65–90.

Another interesting property of a partitioning technique is to evaluate if it is balanced or not. Intuitively, a partitioning is said to be balanced if the obtained splits contain approximately the same number of records.

Definition 6 (*Balanced partitioning*) A partitioning $P = \{p_1, \ldots, p_h\}$ for a dataset D is said to be balanced, with accuracy ε, if and only if:

$$\forall p_i, p_j \in P : abs(|p_i| - |p_j|) \leq \varepsilon \tag{4}$$

where $|p_i|$ denotes the cardinality of the split p_i.

Clearly, the method applied to divide the k-dimensional space, thus producing the k-regions of the splits, has a great impact on the balancing property of the obtained partitioning. The default technique provided by Hadoop, which is based only of the byte-threshold criteria, is able to provide the maximum level of balancing, but it is not context-aware. Conversely, the partitioning techniques described in "Related work" section are context-aware, but when the context attributes are *not* uniformly distributed, they can produce very unbalanced splits, as we will demonstrate in "Case study evaluation" section; this fact can be easily seen in Fig. 2, since the number of points inside each cell is variable. The CoPart partitioning technique proposed in this paper is both balanced and context-aware.

Approaches to context-based partitioning

In order to partition a dataset D, with respect to a context C, different approaches can be applied. Such approaches can be classified into two main families: *multi-dimensional* and *multi-level* partitioning.

Definition 7 (*Multi-dimensional Partitioning*) Given a dataset D with schema S and a context $C = \{c_1, \ldots, c_k\} \subseteq S$. A context-aware partitioning $P_{\mathsf{MD}} = p_1, \ldots, p_h\}$ for D, with respect to C, is said to be *multi-dimensional*, if it divides the k-dimensional space defined by C into k-dimensional cells (generating a k-dimensional grid) and each k-dimensional cell of such grid becomes the k-region of a split of P_{MD}.

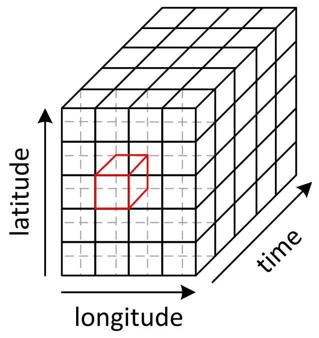

Fig. 3 Example of multi-dimensional partitioning

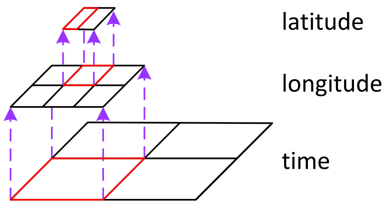

Fig. 4 Example of multi-level partitioning

Figure 3 illustrates an example of multi-dimensional partitioning built w.r.t. three contextual dimensions: 2D space coordinates and time. As you can notice, the partitioning creates a 3-dimensional subdivision of the reference space, the red cube is an example of k-region of a split.

Definition 8 (*Multi-level Partitioning*) Given a dataset D with schema S and a context $C = \{c_1, \ldots, c_k\} \subseteq S$. A context-aware partitioning $P_{\mathrm{ML}} = \{p_1, \ldots, p_h\}$ for D, with respect to C, is said to be *multi-level*, if it recursively divides the reference space by using k mono-dimensional grids, each one corresponding to a dimension in C. In this case, the order of the dimensions has a great impact on the partitioning result.

Figure 4 illustrates an example of multi-level partitioning built w.r.t. three contextual dimensions: 2D space coordinates and time. In this case the following order has been established on the dimensions: time, longitude and finally latitude.

The main difference between these two families resides in the way data can be retrieved using the context attributes. In the case of a multi-dimensional partitioning, we can directly access records by using any or a subset of the context dimensions. Conversely, the second technique implicitly imposes an order among the dimension of analysis. This can be particularly suitable if there is a contextual attribute with a uniform distribution/continuous growing (like for example the temporal dimension), while the other ones are more sparse (i.e., with skewed distribution) inside the reference space.

Let us consider again the example introduced in "Introduction" section. regarding some tourist visits to a predefined set of PoIs, a context $C = \{$ timestamp, latitude, longitude$\}$ and the two partitioning structures illustrated in Figs. 3 and 4. In the case of a selective query involving all the three context attributes, both techniques perform in the same way. On the contrary, in the case of a selective query containing only the PoI longitude (e.g. find all the visits performed on any attraction located on the east of Area), the multi-dimensional partitioning is able to directly select all the cubes with a particular value of longitude, while the multi-level partitioning is not able to perform an initial pruning on the first two levels.

The CoPart technique proposed in this paper belongs to the multi-dimensional partitioning family. In particular, given the main idea to divide the reference space into a set of k-cells, the idea is to build them irregularly (i.e. each cell can have a different shape) and in a way that promotes balancing, even in presence of not uniformly distributed values of the context dimensions.

The identification of the most appropriate way to build the k-cells requires an intuitive and efficient way for evaluating the skewness of the k-points representing the records of D; such evaluation has to be preformed for each context dimension. Based on this evaluation, the right shape of each k-dimensional cell inside the k-dimensional grid can be determined. For this purpose, we extend the idea originally proposed in [9, 10] for the spatial domain to the management of a generic number k of context dimensions. In order to illustrate such idea, we present below the definition of the box-counting function $BC_r^q(D,a)$ for a given dataset D and a context dimension a, that is the fundamental notion that will be applied for obtaining the skewness evaluation of D w.r.t. an analysis dimension a (or simply, a dimension a).

Definition 9 (*Box-counting function for a dimension a*) Given a dataset D, containing an attribute a with values belonging to a domain $\Delta(a)$, and a scale r representing the cell size of a mono-dimensional grid covering the range of values of $\Delta(a)$ appearing in D, the box-counting function $BC_r^q(D,a)$ is defined as:

$$BC_r^q(D,a) = \sum_i \delta_i(D,a)^q \text{with } q \neq 1 \tag{5}$$

where $\delta_i(D,a)$ is the number of records in D whose value for a is contained in the cell i.

The case $q = 1$ is excluded, since $BC_r^q(D,a)$ is equal to the number of records in D, independently from the value of r.

The exponent q allows to take into account different properties of the dataset distribution. Its introduction derives from the concept of fractal dimension D_q, indeed, the box-counting function is part of a technique to compute the fractal dimension of set of points representing fractals or other self-similar shapes. Intuitively, given a grid with cells of side r, the box-counting function with $q = 0$ counts the number of cells that contain at least one record of D. Conversely, when the value of q is greater than 1, the box-counting becomes the sum of the number of records in a cell, raised to q. This function can be used to detect the skewness of a dataset attribute by computing it for $q = 0$ and $q = 2$, while varying the value of r. More specifically, the level of skewness of a dataset attribute, w.r.t. to its reference space, depends on how this value changes while increasing r.

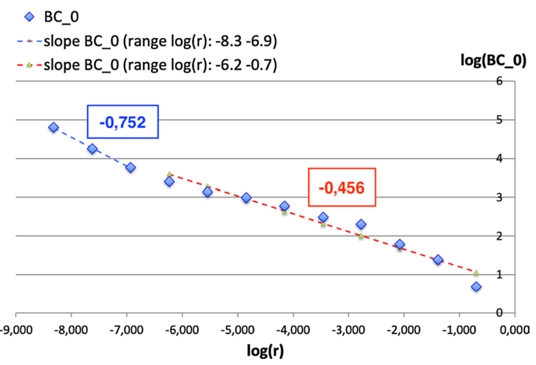

Fig. 5 Example of box-counting plots ($BC_r^0(D, long)$) for the context attribute representing the longitude with $q = 0$. Notice that, in order to extract the behavior of the dataset on the largest scale range, the computation of the slope takes into account the presence of variations in the sequence of values. At the end, the slope of the straight line with maximum support (the red one in this case) is chosen

Definition 10 (*Box-counting plot for a dimension a*) Given a dataset D, containing an attribute a with values belonging to a domain $\Delta(a)$, the box-counting plot of a is the plot of $BC_r^q(D, a)$ versus r in logarithmic scale.

Given the box-counting plot $BC_r^q(D, a)$ for an analysis attribute a in a dataset D, the following observations, extended from the ones in [10, 18], are also valid in a multi-dimensional context and can be used to estimate the distribution of the values of a in D.

Observation 1 *For finite datasets representing fractals and real datasets, the box counting plot reveals a trend of the box counting function that, in a large interval of scale values r,behaves as power law:*

$$BC_r^q(D, a) = \alpha \cdot r^{E_q} \tag{6}$$

where α is a constant of proportionality and E_q is a fixed exponent that characterizes the power law.

Given a dataset D and an attribute a, the power low exponent E_q can be computed by starting from the corresponding box-counting plot. Indeed, E_q corresponds to the slope of the strait line that approximates $BC_r^q(D, a)$ in a range of scale (r_1, r_2); therefore, it can be computed by a linear regression procedure.

Figures 5 and 6 illustrate two examples of box-counting plot, one with $q = 0$ and the other with $q = 2$, for the tourist dataset introduced in "Introduction" section considering the PoI longitude as context dimension. Similar plots can be produced for other dimensions.

Notice that the exponent E_* that is computed by exploiting the Box-counting plot for a dimension a alone, is always a value between 0 and 1, since in this case the reference space that contains the values of a is mono-dimensional. In the general case, the values of E_* belongs to the interval $[0, \ldots, dim(space)]$, where $dim(space)$ represents the dimension of the reference space where the k-points are embedded in (therefore, $dim(space)$=k).

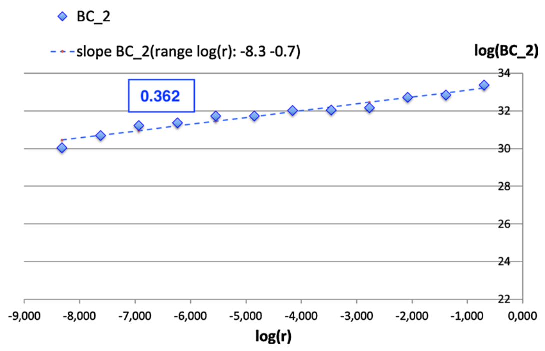

Fig. 6 Example of box-counting plots ($BC_r^2(D, long)$) for the context attribute representing the longitude with $q = 2$

Observation 2 *Given a dataset D and an attribute a characterized by a box-counting function* $BC_r^q(D, a)$*, the exponents* E_0 *and* E_2 *of the corresponding power law could be used as reference descriptors for the distribution of the values of a in D.*

- *The exponent* E_0 *can be used as an indicator of the coverage of the values of a* in *D, namely it allows to identify cases where the dataset leaves empty some areas of the range of values of a covered by D.*
- *The exponent* E_2 *can be used to identify the presence of different concentrations around some values with respect to others, i.e. the situations where there are no empty areas, but different data concentrations in different areas.*

The idea proposed in this paper is to exploit the knowledge about the attribute value distribution, in order to build the best multi-dimensional grid w.r.t. the balancing criteria. In order words, instead of building a *k*-dimensional grid by dividing each space dimension in a uniform way, the division to be applied is identified by considering the attribute value distribution. The idea is to produce cells with potentially different shapes and extensions, but containing a similar number of records.

The following section illustrates in details a MapReduce procedure able to efficiently compute the CoPartpartitioning on a given dataset *D* and a context *C*.

CoPart partitioning technique

Overview

The CoPart partitioning technique proposed in this paper performs a subdivision of a dataset *D* on the basis of a context analysis *C*. More specifically, the *k* dimensions of analysis contained in *C* define a *k* dimensional space, where the records of *D* are located. Such *k*-dimensional space is divided into a set of *k*-dimensional cells, each one representing the *k*-region of a partitioning split, where we will place the intersecting records. The shape and dimension of such cells are determined in accordance with the distribution assumed by the values of the contextual dimensions. The main objective is to guarantee that the partitioning is not only context-aware, but also balanced, namely that produces splits containing a uniform number of records.

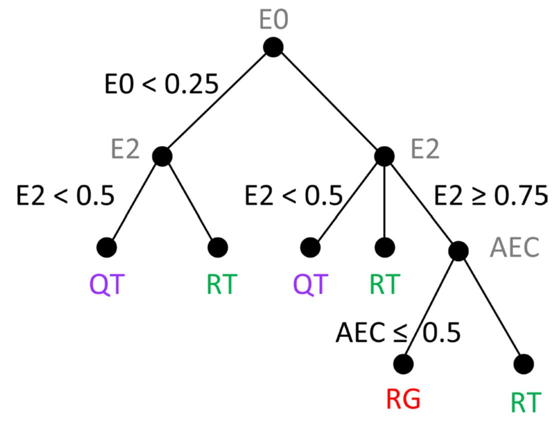

Fig. 7 Decision tree

To evaluate how to best partition the dataset, we need to compute E_0 and E_2 for each contextual dimension. Division techniques can be classified between (a) space-based (e.g. regular grid, kd-trees, or Quad-tree) or (b) record-based (e.g. R-tree). In the case of a space-based solution, the division is obtained starting from the reference space and by recursively dividing it into a fixed number of parts, until a desired threshold is reached (i.e. until all cells contain less than a given number of records). Conversely, in the case of a record-based solution the idea is to start from the data objects and to aggregate nearby ones until a desired threshold is reached. Following the analysis in [10], depending on the values of E_0 and E_2, we adopt a Quad-tree like division (denoted as QT), R-tree like division (denoted as RT), or a uniform division (denoted as RG)—see Fig. 7 for an illustration of the decision tree used by CoPart. In the decision tree each node is labelled by the parameter that is considered to choose the next branch to follow. One of the branches has no condition and it is selected only if all the other conditions are false. The parameters are the computed exponents E_0 and E_2 and an additional parameter that is obtained by analyzing how many cells are found empty during the calculation of the box-counting function. This parameter is called AEC, which stands for average number of empty cells and is used for further refining the choice between the partitioning approaches.

The computation of E_0 and E_2 on a big dataset D, for each contextual dimension $a \in C$, may require many resources. In order to optimize such computation, we propose the following approach: we consider a sample of D (usually 10% of the records is enough [16]) on which we generate a histogram referring to a k-dimensional grid subdividing the space defined by C. This histogram counts the number of records falling inside each k-dimensional cube. The projection of the k-dimensional cubes on each dimension produces k one-dimensional histograms useful for computing the box-counting functions. More specifically, the histogram cells store the values $\delta_i(D, a)$ presented in Eq. 5, from which we can compute the value $BC_r^q(D, a)$. This histogram is built considering the finer grid, i.e. the minimum value of r. The box-counting function for increasing values of r is computed by progressively aggregating the cells of this grid, thus avoiding the explicit computation of the successive histograms on coarser grids. In the next section we present a MapReduce implementation of this technique.

MapReduce implementation of box-counting

In order to speed up the computation of E_0 and E_2 on a big dataset, we propose the MapReduce implementation reported in Algorithms 1–2. More specifically, the map task in Algorithm 1 is responsible for the construction of the k-dimensional histogram on a base grid $grid_0$. The base grid $grid_0$ is created for each mapper during a preliminary setup phase (lines 3–5) by using the dataset MBV (*mbv* in the algorithms) and the initial value r_0; moreover, a map $bcount_0$ is initialized for counting the number of records contained in each cell. In other words, while $grid_0$ is the base k-dimensional grid for the computation of the box-counting function, $bcount_0$ corresponds to its histogram. The values of MBV and r_0 are assumed to be initialized as a configuration parameter (see label **conf**) in line 1. Notice that the notion of MBV and the method intersects have been extended to work on a configurable number of dimensions. Subsequently, during the map phase (lines 6–9), each mapper works on the set of records contained in its assigned split. For each record, the map method determines the set of intersecting cubes (line 7) and updates the corresponding counters in $bcount_0$. A final cleanup method is used by each mapper for summarizing the results produced on its records, producing a portion of the complete histogram. The complete histogram will be computed during the shuffle phase and will be used in the reduce phase for producing the E_0 and E_2 values of each dimension. More specifically, since the key of each map result is the cell, the shuffle will combine all the counters related to the same cell into a list of values. Notice that the use of the map $bcount_0$ allows to represent only the non-empty cubes, potentially reducing the amount of memory required during the computation.

Algorithm 1: Box-counting: map task.

```
1  class Mapper
       conf: $r_0, mbv$
2      $grid_0, bcount_0$
3      method Setup()
4          $grid_0 \longleftarrow$ Grid($r_0, mbv$)
5          $bcount_0 \longleftarrow$ Map()
6      method Map($id$, $record$)
7          $cubes \longleftarrow grid_0$.intersects($record$)
8          foreach $c_j \in cubes$ do
9              $bcount_0.put(c_j, bcount_0.get(c_j) + 1)$
10     method Cleanup()
11         foreach $\langle c_j, p_k \rangle \in bcount_0$ do
12             Write($c_j, p_k$)
```

Algorithm 2: Box-counting: reduce task.

```
class Reducer
    conf: $mbv, r_0, C = \{a_1, \ldots, a_k\}$
    $grids, bcounts, dsSz, spSz, rcSz$
    method Setup()
        $grids, bcounts \longleftarrow \emptyset$
        $r \longleftarrow r_0$
        while $r < mbv.\mathsf{width}()$ do
            $grids.\mathsf{add}(\mathsf{Grid}(r, mbv))$
            for $a \in C$ do
                $bcounts.\mathsf{put}(\langle r, a\rangle, \mathsf{Map}())$
            $r \longleftarrow r * 2$
    method Reduce($c_j, counts$)
        $total \longleftarrow 0$
        foreach $v \in counts$ do
            $total \longleftarrow total + v$
        foreach $i \in [0 \ldots |grids|]$ do
            if $i = 0$ then
                foreach $a \in C$ do
                    $bc_0^a \longleftarrow bcounts.\mathsf{get}(\langle 0, a\rangle)$
                    $bc_0^a.\mathsf{put}(c_j, total)$
            else
                $gr_i \longleftarrow grids.\mathsf{get}(i)$
                $c_k \longleftarrow gr_i.\mathsf{getCube}(c_j)$
                foreach $a \in C$ do
                    $bc_i^a \longleftarrow bcounts.\mathsf{get}(\langle i, a\rangle)$
                    $bc_i^a.\mathsf{put}(c_k, bc_i.\mathsf{get}(c_k) + total)$
    method Cleanup()
        foreach $a \in C$ do
            $BP_0^a, BP_1^a \longleftarrow \emptyset$
            foreach $g_i \in grids$ do
                $bc_i^a \longleftarrow bcounts.\mathsf{get}(\langle i, a\rangle)$
                $BC_a^0 \longleftarrow bc_i^a.\mathsf{size}()$
                $BC_a^2 \longleftarrow 0$
                foreach $\langle c_j, p_k\rangle \in bc_i^a$ do
                    $BC_a^2 \longleftarrow BC_a^2 + p_k^2$
                $BP_0^a.\mathsf{put}(\log(g_i.\mathsf{cellSize}()), \log(BC_a^0))$
                $BP_2^a.\mathsf{put}(\log(g_i.\mathsf{cellSize}()), \log(BC_a^2))$
            $E_0^a \longleftarrow \mathsf{regressionSlope}(BP_0^a)$
            $E_2^a \longleftarrow \mathsf{regressionSlope}(BP_2^a)$
            $aec \longleftarrow$
              $\mathsf{avgEptCells}(g_0, bcounts.\mathsf{get}(\langle 0, a\rangle))$
            $p_a \longleftarrow$ BuildPart($mbv.\mathsf{project}(a)$,
                         $E_0^a, E_2^a, aec, g_0$,
                         $bcounts.\mathsf{get}(\langle 0, a\rangle)$,
                         $dsSz, spSz, rcSz, |C|$)
            Write($a, p_a$)
```

The reduce phase, presented in Algorithm 2, uses the projection of the *k*-dimensional cubes to compute E_0 and E_2 for each contextual dimension, and to determine the best division for its values. More specifically, the computation of the box-counting function requires to consider a set of grids having an increasing dimension of the cell side r, and to maintain a set of counters, one for each cell side of r and contextual dimension $a \in C$.

Given the current grid $grid_i$ with cube side r, we compute the next one, i.e. $grid_{i+1}$, by simply merging a cube with the next one in each dimension, obtaining a new cube with side $2r$. A sequence of considered grids is built in this way during a setup phase (line 3–10). For each of these grids, k corresponding maps, denoted as $bcounts_{\langle i,a \rangle}$, are initialized, each one referring to the projection and aggregation of the histogram *bcounts* w.r.t. the context dimension $a \in C$ and the new cube side r_i. After the initialization, the reduce phase (lines 11–25) is responsible for processing the histogram computed during the map phase and for populating all the other projected and aggregated ones. Each invocation of the reduce method works on a specific cell $c_j \in grid_0$, whose value is a list of counters *counts* computed during the map phase. Inside the reduce phase a first cycle (lines 13–14) is used to sum the values in the list, producing the total counter for c_j. Then the following cycle (lines 15–25) is responsible for populating the corresponding counters in the other grids. The first conditional block (lines 16–19) simply executes the projection of the histogram $bcount_0$ w.r.t. the contextual dimensions a in C. Conversely, the second conditional block (lines 20–25) aggregates the projected values contained in the base grid $grid_0$. The function $gr_i.\mathsf{getCube}(\mathsf{c_j})$ returns for a cube $c_j \in grid_0$ the corresponding cube $c_k \in gr_i$. The cube c_k can be easily computed starting from c_j and the grid level i, which determines the degree of aggregation. Given the cube c_k, the method updates the value of c_k in each map $bc_i^a \in bcounts$ corresponding to the grid gr_i and dimension a. Finally, a cleanup phase (lines 26–44) is used to compute both the values E_0 and E_2 for each contextual dimension $a \in C$ and the best subdivision, given such values. For each grid, the variable BC_a^0 contains the value $BC_r^q(D, a)$ for $q = 0$, while BC_a^2 for $q = 2$. For each contextual dimension a, we build two lists, denoted as BP_0^a and BP_2^a, containing the sequence of pairs $[\langle log(r_i), log(BC_a^0)\rangle, \ldots]$ and $[\langle log(r_i), log(BC_a^2)\rangle \ldots]$, which represents the corresponding box-counting plots (lines 29–36). Finally, for each dimension a, the value of E_0^a and E_2^a, namely the fixed exponents of the computed power laws, is obtained by performing a linear regression of the values in BP_0^a and BP_2^a and by getting the slope of the straight lines (lines 37–38), see function regressionSlope.

In line 39 the average number of empty cells is computed starting from the base grid and the *bcount* map. This value gives a hint about the amount of empty cells contained in the histogram and it is useful as an indicator of the presence of empty zones inside the reference space. Given the computed values E_0^a and E_2^a, for each contextual dimension $a \in C$, together with its MBV, the reducer will finally produce and write the best division for these values (line 40–44), through the function function BuildPart. Notice that $mbv.\mathsf{project(a)}$ computes the projection of the overall dataset MBV w.r.t. the dimension a.

The details of function BuildPart are reported in Algorithm 3. Given the values of E_0 and E_2 for a contextual attribute a, the function uses the heuristics presented in Fig. 7 for selecting the most suitable partitioning technique for it. Moreover, such division can be built using the initial grid $grid_0$ and its counters $bcounts\langle 0, a\rangle$ already computed during the reduce phase. The function requires the following other parameters that are useful for the effective construction of the partitions: the projection of the MBV w.r.t. *a*, namely its range of values in the dataset *D* (see variable *mbv*), the dataset size (see variable *dsSz*), the desired split size (see *spSz*) and the average record size (see *rcSz*).

Algorithm 3: BuildPart

```
1  BuildPart($mbv, E_0, E_2, aec, gr_0, bc, dsSz, spSz, rcSz, k$)
2      $splits \longleftarrow dsSz/spSz$
3      $divs \longleftarrow \sqrt[k]{splits}$
4      $\theta \longleftarrow \lfloor spSz/rcSz \rfloor$
5      $side \longleftarrow mbv.\mathsf{width}/divs$
6      if $E_0 < 0.25$ then
7          if $E_2 < 0.50$ then
8              Qt($mbv.min, mbv.max, gr_0, bc, side, \theta$)
9          else
10             Rt($mbv.min, mbv.max, gr_0, bc, side, \theta$)
11     else
12         if $E_2 \geq 0.75$ then
13             if $aec \leq 0.5$ then
14                 Rg($mbv.min, mbv.max, side$)
15             else
16                 Rt($mbv.min, mbv.max, gr_0, bc, side, \theta$)
17         else if $E_2 < 0.50$ then
18             Qt($mbv.min, mbv.max, gr_0, bc, side, \theta$)
19         else
20             Rt($mbv.min, mbv.max, gr_0, bc, side, \theta$)
```

The first operation performed by BuildPart is the computation of an indicative side length for the partitioning of *a*. This is done in lines 2–5, where the global number of required splits is stored in *splits*, from this value the number of division for each dimension is given by the computing the *k*th-square, where *k* is the number of considered dimensions. Finally, the indicative side length is computed and stored in *side* as the ratio between the MBR width and the number of divisions. Based on the given E_0 and E_2 values, the function simply invokes the construction of the right partitioning for current contextual dimension *a*.

Partitioning techniques

We consider three kinds of partitioning: regular grid (Rg), space-based or Quad-tree like (Qt), and record-based or R-tree like (Rt).

The construction of the space-based partitioning (Qt) is illustrated in Algorithm 4. The algorithm returns a list of partitions or divisions for the values of the dimension *a*, represented by the variable *part* which contains the boundaries of such divisions. The extent of the reference space for the values of *a* is represented by the input variables *or* (i.e., origin) and *en* (i.e., end).

Algorithm 4: QT: QuadTree construction

```
Qt($or, en, gr_0, bc_0, side, spSz$)
    $tot \longleftarrow 0$
    $part \longleftarrow \{or\}$
    $p \longleftarrow or$
    $i \longleftarrow$ firstCell($or, side$)
    while $tot < \theta \wedge p < en$ do
        $c \longleftarrow gr_0$.get($i$)
        $tot \longleftarrow tot + bc_0$.get($c$)
        $p \longleftarrow p + side$
        $i \longleftarrow i + 1$
    if $p \geq en$ then
        $part \longleftarrow part \cup \{en\}$
    else if $p \leq en/2$ then
        $part \longleftarrow$ Qt($or, en/2, gr_0, bc_0, side, \theta$)
        $part \longleftarrow part \cup$
          Qt($or + en/2, en, gr_0, bc_0, side, \theta$)
    else
        $part \longleftarrow part \cup$
          Qt($or + en/2, en, gr_0, bc_0, side, \theta$)
    return $part$
```

The main idea underlying the function QT is to build the partitions by aggregating the cells of gr_0 inside the reference space [*or*, *en*] until a given threshold is reached (lines 6–10). Such threshold represents the maximum number of records per cells and can be compared with the sum of counters in bc_0 for the aggregated cells. The aggregation stops if the threshold has been reached ($tot < \theta$) or the end of the reference space has been reached ($p < en$). Reaching the end represents the base case of the recursion, all the cells of gr_0 inside the reference space have been successfully aggregated without violating the threshold constraint. Therefore, the end value *en* can be added to the partition list and the function can terminate (lines 11–12). Conversely, reaching the threshold represents the recursive case, in this case the considered reference space (represented by the interval [*or*, *en*]) has to be recursively split by 2 in an attempt to build a partition that does not violate the threshold constraint. In this case two situations have to be distinguished, (1) if the aggregation reaches the threshold before reaching the middle of the reference space, both middle parts need to be split in a more dense way (see the recursive calls at lines 13–16). Conversely, (2) if the first middle of the reference space can be aggregated without reaching the threshold, the recursive call is needed only on the second middle (lines 17–19)

The construction of an R-tree like division is illustrated in Algorithm 5. It proceeds in a way very similar to the previous one by aggregating the cells of gr_0 until a given threshold θ is exceeded or the end of the reference space *en* is reached (lines 7–11). However, in this case the reference space (between *or* and *en*) does not require to be recursively divided, but can be freely aggregated. Therefore, reaching *en* represents the base case (lines 12–13), while the exceeding of the threshold θ involves a recursive call on a smaller reference space (lines 15–16).

Algorithm 5: RT: R-Tree construction

```
1  method $\textsc{Rt}(or, en, gr_0, bc_0, side, \theta)$
2      $tot \longleftarrow 0$
3      $part \longleftarrow \{or\}$
4      $p \longleftarrow or$
5      $f \longleftarrow \mathsf{T}$
6      $i \longleftarrow \mathsf{firstCell}(or, side)$
7      while $tot < \theta \wedge p < en$ do
8          $c \longleftarrow gr_0.get(i)$
9          $tot \longleftarrow tot + bc_0.get(c)$
10         $p \longleftarrow p + side$
11         $i \longleftarrow i + 1$
12     if $p \geq en$ then
13         $part \longleftarrow part \cup \{p\}$
14     else
15         $part \longleftarrow part \cup$
16             $\textsc{Rt}(or + (side * i), en, gr_0, bc_0, side, \theta)$
17     return $part$
```

The last kind of splitting approach is represented by the regular grid (Rg) illustrated in Algorithm 6. In this case, given the fixed cell length *side*, computed during the function BuildPart, function Rg simply divides the reference space in homogeneous parts having such fixed dimension (lines 4–6). Only the last split can have a length smaller than *side*.

Algorithm 6: RG: Regular grid construction

```
1  $\textsc{Rg}(or, en, side)$
2      $part \longleftarrow \{\}$
3      $p \longleftarrow or$
4      while $p < en$ do
5          $part \longleftarrow part \cup \{p\}$
6          $p \longleftarrow p + side$
7      $part \longleftarrow part \cup \{en\}$
8      return $part$
```

All these functions are useful for computing the output of the BoxCounting job in Algorithms 1–2.

CoPart

The list of k 1-dimensional grids, one for each context dimension, is used by the CoPart job that actually performs the partitioning of the input dataset. The details of CoPart are illustrated in Algorithm 7. The map task receives, as a configuration parameter, the k-dimensional partitioning grid built combining the previously computed k divisions (line 1). The map task works on each record of its split by determining the list of intersecting cubes (line 3), then for each of them it writes in output the pair composed by the cube identifier, as key, and the record itself, as value (line 5). Many reducer can be instantiated, one for each cube of *part*: indeed, each reducers can work on a cube at time by simply writing in output the records contained in this cube (line 9). The output of each reduce tasks becomes a split on the HDFS.

Algorithm 7: CoPart job

```
1 class Mapper
      conf: $part$
2     method Map($id$, $record$)
3         $cubes \longleftarrow part$.intersects($record$)
4         foreach $c \in cubes$ do
5             Write($c, record$)
6 class Reducer
7     method Reduce($c_i, \{r_1, \ldots, r_{n_i}\}$)
8         foreach $r_j \in \{r_1, \ldots, r_{n_i}\}$ do
9             Write($r_j$)
```

Notice that in order to make the partitioning effective, we also build a *master file*, like the one provided by SpatialHadoop, which contains the list of *k*-dimensional splits together with their boundaries and the name of the split file containing its intersecting records.

Case study evaluation

Perfomance metrics

We present a set of quality metrics that may be used for evaluating and comparing different context-based partitioning techniques. In particular, such quality measures have been inspired and extended from the ones proposed in [16], some of which have been derived from the R*-tree optimization criteria and related to the performance of the range query [6].

Definition 11 (*Q1—Total Volume*) Given a context-based partitioning *P* composed by a set of *k*-dimensional cubes $\{c_1, \ldots, c_n\}$, Q1 measures the total volume occupied by all cubes:

$$Q1 = \sum_{i=1}^{n} \prod_{j=1}^{k} |c_i.mbv[j].\max - c_i.mbv[j].\min|$$

where $c.mbv[j]$ is the projection of the *k*-dimensional MBV of *c* w.r.t. the dimension *j*, while $mbv[j].\min$ and $mbv[j].\max$ returns respectively the minimum and maximum value in the MBV range for dimension *j*.

Q1 can be used as an indicator of the *dead space* covered by partitions without containing any actual data.

Definition 12 (*Q2—Total Overlap*) Given a context-based partitioning P composed by a set of k-dimensional cubes $C = \{c_1, \ldots, c_n\}$, Q2 measures the total overlap between pairs of partitions.

$$Q2 = \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{m} \mathsf{insersection}(\mathsf{c_i.volume}, \mathsf{c_j.volume})$$

where $c.\mathsf{volume}$ computes the volume of k-dimensional cube c.

Definition 13 (*Q3—Total Margin*) Given a context-based partitioning P composed by a set of k-dimensional cubes $C = \{c_1, \ldots, c_n\}$, Q3 measures the total margin of all partitions, where the margins of a cube is the sum of its dimension lengths.

$$Q3 = \sum_{i=1}^{n} \sum_{j=1}^{k} |c_i.mbv[j].\mathsf{max} - \mathsf{c_i.mbv[j].min}|$$

Definition 14 (*Q4 Load Balance*) Given a context-based partitioning P composed by a set of k-dimensional cubes $C = \{c_1, \ldots, c_n\}$, Q4 is the average standard deviation of the partition sizes.

$$Q4 = \mathsf{avg}(\mathsf{std}(\mathsf{c_1.size, C}), \ldots \mathsf{std}(\mathsf{c_n.size, C}))$$

where $c.\mathsf{size}$ returns the size of the split c, while $\mathsf{std}(\mathsf{c.size}, C)$ computes the standard deviation of the size of c w.r.t. the mean size of the splits in C, and $\mathsf{avg}(\mathsf{v_1}, \ldots, \mathsf{v_n})$ returns the average between the provided values.

Q4 is particularly useful for evaluating the balancing of the partitioning technique at hand.

Dataset

The proposed technique has been applied to a real-world dataset containing the swipes of a city pass, called *VeronaCard*, which is provided by the tourist office of Verona, a municipality in Northern Italy. The dataset contains about 24,000,000 records concerning 10 years. Each record reports beside to the identifier of the city pass and the name of the visited PoI: the location (coordinates) of the PoI, the entrance timestamp and the age of the tourist holding the card. More specifically, the original dataset has been properly perturbated in order to simulate different distributions on each dimensions, as reported in Table 1.

Experimental methodology

The performances of the proposed technique have been evaluated with respect to both the quality metrics reported in "Perfomance metrics" section and the range query operation. In particular, we compare the proposed technique with the traditional multi-dimensional partitioning technique (MD) presented in Definition 7 and the default random partitioning (RP) provided by Hadoop, which represents the baseline for the results. We provide a MapReduce implementation of the multi-dimensional partitioning,

Table 1 Datasets characteristics

Ds	Dim	E_0	E_2	%EC	Part	#Div
D_1	Lat	0.952	0.922	0.842	RT	3
	Long	0.993	0.952	0.804	RT	3
	Time	1.000	0.977	0.778	RT	3
	Age	0.500	0.450	0.982	QT	5
D_2	Lat	0.992	0.997	0.807	RT	4
	Long	1.000	0.965	0.778	RT	3
	Time	1.000	0.977	0.778	RT	3
	Age	0.000	0.000	0.997	QT	8
D_3	lat	0.707	0.515	0.978	RT	3
	Long	0.456	0.362	0.991	QT	6
	Time	1.000	0.977	0.778	RT	3
	Age	0.500	0.450	0.982	QT	4

Table 2 Partitioning characteristics

Ds		RP	MD		CoPart	
	Size (Gb)	#blks	#cells	#blks	#cells	#blks
D_1	2.85	23	3	24	108	108
D_2	2.83	23	3	24	198	198
D_3	2.85	23	3	24	120	120

the CoPart technique and the multi-dimensional range query[1]. The characteristics of the considered datasets are reported in Table 1 where columns E_0, E_2 and %EC contain the values of the exponents E_0, E_2 (obtained by applying algorithms of "MapReduce implementation of box-counting" section) and the percentage of empty cells for each dimension, column Part is the corresponding partitioning technique applied for that dimension by CoPart, whereas #Div is the number of obtained subdivisions.

Some metadata about the considered partitioning techniques are reported in Table 2, where #blks is the number of HDFS blocks (or splits) and #cells is the number of index cells. As you can notice, for the random partitioning, we have not specified any number of cells, since it does not build an index, while the number of cells for MD is very small w.r.t. to the number of splits. Indeed, in some cases a single cell contains almost all the records and it has been physically subdivided into many other splits to fit the HDFS block size. Notice that, in the MD technique the subdivision of the space generates k-dimensional cubes from regular cells in each dimension. In other words, it computes the number of required splits n_s according to the dataset size and split size, then the cube with the smallest number of cells that is greater than n_s is generated. In the considered case the cube has 81 cells, but only three of them contain all the dataset records. Conversely, with CoPart, the content of each cell can fit into a single physical split. Moreover, the number of cells

[1] https://github.com/smigliorini/context-partitioning.

Table 3 Quality metrics for the various partitioning techniques

Ds	Partitioning					
	MD			CoPart		
	Q1	Q3	Q4 (%)	Q1	Q3	Q4 (%)
D_1	2.68E10	1.25E12	173	1.29E10	2.27E13	67
D_2	1.63E10	1.26E12	173	2.18E09	4.16E13	59
D_3	2.20E10	1.25E12	173	1.97E09	2.52E13	68

effectively stored by CoPart could be different from the Cartesian product of the number of subdivisions, since empty partitions are discarded.

For each dataset we randomly generate a set of range queries in the following way: we consider an increasing overlapping w.r.t. the reference space from 5 to 95% and for each of them we randomly produce 10 range windows with that percentage of overlapping. Given them, we apply our multi-dimensional range query operation on the random partitioning, the multi-dimensional partitioning and the CoPart.

Results

The results in terms of quality metrics are reported in Table 3. Q1 is an indicator of the dead space, namely the space covered by partitions without containing any actual data. Clearly, less is the value of Q1 better is the index. As you can notice the CoPart partitioning provides smaller values than the MD technique, in some cases even an order of magnitude smaller. Q2 is not considered because both indexes produce partitions that do not overlap. Q3 is the total margin, so greater is the value, better is the index: in all cases the CoPart technique have values that are an order of magnitude better. Finally, Q4 is a measure of balancing, a smaller value means that the difference between the partition sizes is smaller. The CoPart technique is able to create more balanced partitions w.r.t. the MD one.

The results in terms of range query execution are reported in Table 4 where: column *DS* contains the dataset name, *OV* is the percentage of overlap of the query window w.r.t. the reference space, RP, MD and CoPart denote the three partitioning techniques and for each of them *s* and *maps* denote the average time in seconds and the average number of maps obtained in the 10 queries, respectively. The experiments have been performed on a Hadoop-based cluster composed of 10 nodes, each one with 8GB of RAM and 4 cores.

Finally, Table 5 summarizes the time required by the MD and CoPart technique for effectively perform the partitioning of the three datasets. In particular, beside to the time required by the two techniques (*total time*), for the CoPart technique we distinguish between the time required for computing the box counting, namely identifying the partitioning grid (*bc s*), from the time required for effectively partitioning the data (*part s*) given the multidimensional grid. Clearly the random technique is not reported, since it is applied by default by Hadoop during the loading of the data inside the HDFS.

Table 4 Results of the range query experiments

DS	OV (%)	RP		MD		CoPart	
		s	Maps	s	Maps	s	Maps
D_1	5	364	23	85	5	27	2
	10	364	23	121	7	30	3
	15	364	23	234	14	80	11
	25	370	23	299	18	122	15
	30	380	23	310	18	151	20
	50	390	23	390	23	265	34
	75	402	23	402	23	299	38
D_2	5	368	23	86	5	26	2
	10	367	23	120	7	31	3
	15	377	23	236	14	99	16
	25	370	23	299	18	147	22
	30	378	23	307	18	172	26
	50	388	23	389	23	350	55
	75	401	23	402	23	380	58
D_3	5	374	23	85	5	23	1
	10	374	23	187	7	26	2
	15	381	23	235	14	85	9
	25	379	23	301	18	60	6
	30	388	23	275	16	127	14
	50	426	23	415	23	219	24
	75	437	23	427	23	251	27

Table 5 Time required for performing the partitioning of the three datasets with MD and CoPart techniques

Dataset	MD	CoPart		
	Total s	Bc s	Part s	Total s
D_1	87	42	94	136
D_2	90	40	92	132
D_3	93	48	97	145

Discussion

The experimental results shown in Table 3 confirm that the CoPart technique produces more balanced partitions w.r.t. the MD one in relation to any of the considered quality metrics. Moreover, from the results reported in Table 4, we can notice that: (a) the RP technique requires to always load all the splits, independently from the size of the range query, since no spatial criteria have been applied for the partitioning. Moreover, the time required for performing the range query does not change so much, since the number of comparisons to be performed is essentially the same in all cases. (b) The filtering capacity of the MD partitioning is not so high, starting from a query window with an overlap of 25%, almost all the splits have to be processed. (c) Finally, the CoPart technique is able to perform a very selective filtering on the input data: the number of considered splits increases with the size of the query window.

Moreover, even if the number of map tasks to be instantiated is greater than the ones of MD (due to the greater number of cells), the performances of the range query are significantly improved in any experiment. This is due to the fact that even if MD produces a smaller number of splits, as described in Table 2, we have a cell containing almost all the data. Therefore, the map task which processes such split will do almost all the work, considerably decreasing the positive effect of a parallel computation. Moreover, the pruning capabilities related to the application of a partitioning technique is very limited due to the skewness of the original datasets which is not properly captured by a uniform subdivision.

Another aspect to be considered is the overhead induced by the application of the partitioning technique. Any partitioning technique requires at least to rewrite the entire dataset in order to obtain splits reflecting the chosen subdivision criteria. Therefore, the time required to complete such task increases as the size of the dataset increases. Anyway, the general idea is that this cost is compensated by the pruning capabilities induced by the new organization and is as greater as the dataset is bigger and the query is more selective. Moreover, as mentioned in the introduction the application of a partitioning technique is as more justified as the number of queries or analysis performed on the same dataset increases. In the case of occasional queries performed once and never more, the overhead induced by a preliminary partitioning could not be justified.

Given such considerations, the time reported in Table 5 for partitioning datasets D_1 $-D_3$ has to be actually considered as a common overhead for all the performed range queries together, not a single overhead added to each single operation. In any case, even if we add the time required for the partitioning with the average time required for performing a single query in the various cases, what we can observe is that when the query is more selective, the total time required by MD or CoPartremains sensibly smaller w.r.t. the time required by RP. Conversely, when the query becomes less selective (i.e., almost all the splits have to be processed), the pruning ability of the partitioning technique is less and the total time could become even greater than the time required by the RP approach. See for instance the last row related to each dataset (overlap equal to 75%): in this case both RP and MD process all the splits and the running time for the queries is almost the same; therefore, the partitioning not only introduces no positive effect but could even decrease the overall performances, due to the preliminary required operations.

The last aspect to be considered is the different amount of time required for performing a MD partitioning w.r.t. the CoPart one. Clearly, the first one requires less time, because it simply subdivides the records by using a uniform grid which can be computed in almost a constant time. Conversely, the CoPart technique applies a preliminary step for identifying the distribution of the records w.r.t. the various dimensions and determining the best grid. Indeed the time required by CoPart is quite double the time required by MD, but the performance improvements that it induces is bigger in any case and justifies such additional cost, particularly when it can be amortized by several subsequent queries.

Finally, the last observation that we can add about the partitioning time is that in many practical cluster configurations, included the one used in this paper, the cost of I/O operations is bigger than the CPU one, particularly as regards to the network I/O. Therefore, when we simply shift from an Hadoop cluster to a Spark one, which sensitively reduces the number of intermediate I/O operations, the cost of the partitioning task could be naturally reduced. Indeed, as discussed in the introduction, the proposed partitioning technique could be applied also in the Spark context in order to properly subdivide data into RDDs without any intermediate I/O operations.

Related work

Horizontal partitioning techniques has been widely studied in databases especially in relational and shared-nothing systems [14, 24, 29]; these techniques are adopted to split rows for different purposes, e.g. load balancing, but mainly to improve query processing, since they avoid to load unnecessary blocks of tuples.

The big data era has brought new challenges, therefore the problem has been recently investigated on graph based models, e.g. ontologies, and for NoSQL systems. Ontologies are often used as meta-models to overcome the heterogeneity of different sources. In [27] the authors deal with the problem of computing on-the-fly matching of large-scale ontologies by partitioning them into smaller subsets. In this scenario, Hadoop and the MapReduce paradigm can be used to parallelise the matching phase, but, in order to avoid unnecessary matching between two sub-ontologies, the searching space has to be reduced with a clustering approach, i.e. two graphs are considered similar, and thus stored in the same cluster, on the base of similarity measures working on paths between different nodes.

MapReduce frameworks divide datasets into independent chunks in order to process data in parallel, but they consider only the amount of bytes of each chunk as splitting criteria. To overcome this limitation, some proposals have investigated the problem of fragmenting datasets by using K-Means Clustering based on the analysis of log files [5].

In [28] the authors propose a technique based on frequent itemsets mining, to Partition, Bucket and Sort the Tables of a big data warehouse with the most frequent predicate attributes in the queries: they apply data mining algorithms on a queries workload to determine the most frequent predicate attributes, and use a hash-partitioning technique without making any assumptions about the filters used in the query predicates (i.e. in the Where clause of a SQL query). The contribution addresses data warehouses partition and once the attributes to use for partitioning are mined, they are not combined to obtain a multi-level partitioning.

In [21] the authors investigate the development of an Hadoop-based framework for the parallel construction of a B+-tree with the aim to reduce the data processing time. While a B+-tree supports only mono-dimensional index construction, some multi-dimensional extensions are available in literature. For instance, other previous works propose partitioning approaches mainly based on spatial [17, 31] and spatio-temporal [3, 4] characteristics. In case of spatial partitioning, one may partition based on space (grid and Quad-tree), based on data (STR, STR+, K-d tree), or based on space filling curves (Z-curve, Hilbert curve) [16]. The selection of the partitioning technique is usually left to the user, and only few works automatically select the best partitioning technique based on the dataset distribution [9].

ST-Hadoop [3] considers the spatio-temporal dimensions for the partitioning; each dimension is considered independently, thus the result is a *multi-level partitioning*. In particular, ST-Hadoop [3] firstly divides the dataset based on temporal granularity, and then splits each portion based on spatial proximity. A query focused on spatial properties (e.g., Has PoI *x* been visited more than PoI *y*?) requires the analysis of all, temporally organized, splits.

HadoopTrajectory [4] considers the spatio-temporal dimensions together and builds a *multi-dimensional partition* in such a way that partitions are 3D cubes, where the three dimensions are space (planar coordinates) and time. Given a query focused on one dimension, this approach allows the exact selection of the splits that could be useful in answering the query. The challenge imposed by the multi-dimensional partitioning is to find the best size of the grid cells in each dimension, so that the amount of data in each cell is balanced. This can be a non trivial task, especially in the general case where data are not uniformly distributed [7, 9].

In this paper we generalize the proposals in [3, 4] by observing that each relevant dimension, called in the paper contextual dimension, can be used, similarly to spatial and temporal coordinates, to partition a multi-dimensional space.

There exist other contexts where partitioning is important, such as the case of Edge Computing [20]. The techniques developed there could be considered orthogonal to our approach, since we are focusing on partitioning the data for a more efficient computation, while edge computing subdivides the computation among different nodes.

Conclusion

In MapReduce frameworks the partitioning of a dataset into independent splits is a critical operation, since the degree of parallelism and the overall performances directly depend from the initial partitioning technique. This is particularly true in case of context-based applications, where data present correlations and consequently they could be aggregated and filtered in order to reduce the amount of work to be done during the analysis. Moreover, beside the need for a context-based partitioning technique, in order to produce balanced splits, it is necessary to consider the distribution of the dataset w.r.t. the analysis dimensions.

This paper proposes a context-based partitioning technique which takes care of the data distributions in the analysis dimensions to produce the best partitioning for the dataset at hand. We also apply the proposed technique to a real-world dataset and compare its performances w.r.t. existing partitioning techniques for highlighting its differences and benefits. The results confirm the goodness of the approach and encourage further research in this direction, for instance as regards to the management of multi-accuracy data [8].

As future work, we plan to apply the proposed technique to other operations, like the join one. The join operation would certainly be an interesting application and extension, because in this case each dataset could have its specific distribution and will be partitioned in a different way. The use of a context-based partitioning technique can improve the performance of the join since, instead of considering the Cartesian product of all the possibile pairs of splits, it allows to prune some combinations that surely will not participate to the result, i.e. those having an empty MBV intersection. Moreover, we plan to extend our technique in such a way to include categorical attributes in the notion of context, i.e. attributes whose domains have not a predefined order relation between values and thus a distance between a pair of values cannot be computed. As suggested in [22], in this case it is possible to mine frequent correlated values in order to infer the so called contextual similarity; such measure can be used to store on the same partition similar (i.e. frequent correlated) values.

Acknowledgements
Not applicable.

Authors' contributions
All the authors have contributed equally to this work. All authors read and approved the final manuscript.

Funding
This work was partially supported by the Italian National Group for Scientific Computation (GNCS-INDAM) and by "Progetto di Eccellenza" of the Computer Science Dept., Univ. of Verona, Italy.

Availability of data and materials
The datasets and the source code are freely available at https://github.com/smigliorini/context--partitioning.

Ethics approval and Consent to participate
This manuscript uses a real-world dataset containg the swipes of a city pass, called VeronaCard. Such data have been collected and provided by the tourist office of Verona, a municipality in Northen Italy. The used dataset contains only aggregated and anonymous information which cannot be directly associated to any specific tourist.

Consent for publication
Not applicable. The manuscript does not contain data which is traceable to any individual person.

Competing interests
No financial and non-financial competing interests exist regarding this work.

Received: 8 October 2020 Accepted: 4 January 2021
Published online: 19 January 2021

References

1. White T. Hadoop: the definitive guide. 4th edn. O'Reilly Media, Inc.; 2015.
2. Chambers B, Zaharia M. Spark: the definitive guide big data processing made simple. 1st ed. O'Reilly Media, Inc.; 2018.
3. Alarabi L, Mokbel MF, Musleh M. ST-Hadoop: a MapReduce framework for spatio-temporal data. GeoInformatica. 2018;22(4):785–813.
4. Bakli M, Sakr M, Soliman TH. HadoopTrajectory: a Hadoop spatiotemporal data processing extension. J Geogr Syst. 2019;21(2):211–35.
5. Beck M, Hao W, Campan A. Accelerating the mobile cloud: using amazon mobile analytics and k-means clustering. In: 2017 IEEE 7th annual computing and communication workshop and conference (CCWC); 2017. p. 1–7.
6. Beckmann N, Kriegel HP, Schneider R, Seeger B. The r*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec. 1990;19(2):322–31. https://doi.org/10.1145/93605.98741.
7. Belussi A, Carra D, Migliorini S, Negri M, Pelagatti G. What makes spatial data big? A discussion on how to partition spatial data. In: 10th international confernece on geographic information science (GIScience 2018); 2018, p. 2:1–5.
8. Belussi A, Migliorini S. A framework for integrating multi-accuracy spatial data in geographical applications. Geoinformatica. 2012;16(3):523–61.
9. Belussi A, Migliorini S, Eldawy A. Detecting skewness of big spatial data in SpatialHadoop. In: Proceedings of the 26th ACM SIGSPATIAL international confernce on advances in geographic information systems; 2018, p. 432–5.
10. Belussi A, Migliorini S, Eldawy A. Skewness-based partitioning in spatialHadoop. ISPRS Int J Geo-Inf. 2020;9(4):201. https://doi.org/10.3390/ijgi9040201.
11. Belussi A, Migliorini S, Negri M, Pelagatti G. Validation of spatial integrity constraints in city models. In: Proceedings of the 4th ACM SIGSPATIAL international workshop on mobile geographic information systems; 2015, p. 70–9.
12. Bolchini C, Quintarelli E, Tanca L. CARVE: context-aware automatic view definition over relational databases. Inf Syst. 2013;38(1):45–67.
13. Brézillon P, Abu-Hakima S. Using knowledge in its context: report on the IJCAI-93 workshop. AI Mag. 1995;16(1):87–91. https://doi.org/10.1609/aimag.v16i1.1127.
14. Curino C, Zhang Y, Jones EPC, Madden S. Schism: a workload-driven approach to database replication and partitioning. In: Proceedings of the VLDB endow. 2010; 3(1): 48–57. https://doi.org/10.14778/1920841.1920853. http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R04.pdf.
15. Egenhofer MJ, Franzosa R. Point-set topological spatial relations. Int J Geogr Inf Syst. 1991;2(5):161–74.
16. Eldawy A, Alarabi L, Mokbel MF. Spatial partitioning techniques in SpatialHadoop. Proc VLDB Endow. 2015;8(12):1602–5. https://doi.org/10.14778/2824032.2824057
17. Eldawy A, Mokbel MF. SpatialHadoop: a mapreduce framework for spatial data. In: 2015 IEEE 31st international conference on data engineering; 2015, p. 1352–63.
18. Faloutsos C, Seeger B, Traina A, Traina C Jr. Spatial join selectivity using power laws. SIGMOD Rec. 2000;29(2):177–88.
19. Hashem IAT, Yaqoob I, Anuar NB, Mokhtar S, Gani A, Khan SU. The rise of "big data" on cloud computing: review and open research issues. Inf Syst. 2015;47:98–115. https://doi.org/10.1016/j.is.2014.07.006.

20. Huh JH, Seo YS. Understanding edge computing: engineering evolution with artificial intelligence. IEEE Access. 2019;7:164229–45.
21. Huynh CV, Huh J. B+-tree construction on massive data with hadoop. Clust Comput. 2019;22(Suppl 1):1011–21. https://doi.org/10.1007/s10586-017-1183-y.
22. Ienco D, Pensa RG, Meo R. Context-based distance learning for categorical data clustering. In: Adams NM, Robardet C, Siebes A, Boulicaut J, editors. Advances in intelligent data analysis VIII, 8th international symposium on intelligent data analysis, IDA 2009, Lyon, France, August 31–September 2, 2009. proceedings, *Lecture Notes in Computer Science*, vol. 5772. Berlin: Springer; 2009. , p. 83–94. https://doi.org/10.1007/978-3-642-03915-7_8.
23. Jacobs A. The pathologies of big data. Commun ACM. 2009;52(8):36–44. https://doi.org/10.1145/1536616.1536632.
24. Kumar KA, Quamar A, Deshpande A, Khuller S. SWORD: workload-aware data placement and replica selection for cloud data management systems. VLDB J. 2014;23(6):845–70. https://doi.org/10.1007/s00778-014-0362-1.
25. Migliorini S, Belussi A, Negri M, Pelagatti G. Towards massive spatial data validation with SpatialHadoop. In: Proceedings of the 5th ACM SIGSPATIAL international workshop on analytics for big geospatial data; 2016, p. 18–27.
26. Migliorini S, Belussi A, Quintarelli E, Carra D. A context-based approach for partitioning big data. In: Proceedings of the 23nd international conference on extending database technology, EDBT 2020; 2020, p. 431–4. OpenProceedings.or. https://doi.org/10.5441/002/edbt.2020.50.
27. Mountasser I, Ouhbi B, Frikh B. Hybrid large-scale ontology matching strategy on big data environment. In: Anderst-Kotsis G, editor. Proceedings of the 18th international conference on information integration and web-based applications and services, iiWAS 2016, Singapore, November 28–30. New York: ACM; 2016, p. 282–7. https://doi.org/10.1145/3011141.3011185.
28. Ramdane Y, Boussaid O, Kabachi N, Bentayeb F. Partitioning and bucketing techniques to speed up query processing in spark-sql. In: 2018 IEEE 24th international conference on parallel and distributed systems (ICPADS); 2018, p. 142–51.
29. Sun L, Franklin MJ, Krishnan S, Xin RS. Fine-grained partitioning for aggressive data skipping. In: Dyreson CE, Li F, Özsu MT, editors. International conference on management of data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014. New York: ACM; 2014, p. 1115–26. https://doi.org/10.1145/2588555.2610515.
30. Wu X, Zhu X, Wu G, Ding W. Data mining with big data. IEEE Trans Knowl Data Eng. 2014;26(1):97–107. https://doi.org/10.1109/TKDE.2013.109.
31. Yu J, Zhang Z, Sarwat M. Spatial data management in apache spark: the geospark perspective and beyond. Geoinformatica. 2019;23(1):37–78.

Publisher's Note